A Human-in-the-loop Approach to Generate Annotation Usage Rules

A Case Study with MicroProfile

Mansur Gulami, Ajay Kumar Jha, Sarah Nadi, Karim Ali
{gulami,ajaykum1,nadi,karim.ali}@ualberta.ca
University of Alberta
Edmonton, Canada

Yee-Kang Chang, Emily Jiang
yeekangc@ca.ibm.com,emijiang@uk.ibm.com
IBM
Markham, Canada

Abstract

Frameworks and libraries provide functionality through Application Programming Interfaces (APIs). Developers might misuse these APIs, because their usage rules are often implicit, undocumented, or not readily available in the form of checkable rules. At the same time, manually writing usage rules for each API is time consuming. Therefore, researchers have proposed various techniques to automatically mine API usage rules. However, mined rules are not always accurate, resulting in false positives when used for misuse detection. To overcome these trade-offs, we combine rule mining and manual rule authoring approaches by creating a human-in-the-loop API usage rule generation pipeline. Based on our industrial collaborator's needs, our work focuses on generating annotation-based API usage rules for MicroProfile, a framework designed for building microservices using Enterprise Java. We use a frequent-itemset based pattern-mining technique to mine MicroProfile annotation usage rules and design a GUI-based rule validation tool (RVT) that allows experts to browse through the mined rules to validate (accept, edit, discard) them. Our pipeline then automatically generates checkable API usage rules from the confirmed rules, which can then be used to detect misuses or to enhance documentation. To assess the usefulness of having mined rules as a starting point for rule authoring and to assess the usability of RVT in validating rules, we perform a user study with MicroProfile API experts.

Keywords

Annotations, MicroProfile, API usage rules, API misuse

ACM Reference Format:
Mansur Gulami, Ajay Kumar Jha, Sarah Nadi, Karim Ali and Yee-Kang Chang, Emily Jiang. 2022. A Human-in-the-loop Approach to Generate Annotation Usage Rules: A Case Study with MicroProfile. In *Proceedings of Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering (CASCON'22).* ACM, New York, NY, USA, 10 pages.

CASCON'22, November 15–17, 2022, Toronto, Canada


```
1 @Liveness
2 public class AuthServiceHealthCheck implements HealthCheck {
3     @Override
4     public HealthCheckResponse call() { /*omitted*/ }
5 }
```

Figure 1: An example illustrating the usage of the MicroProfile `@Liveness` annotation.

1 Introduction

Java annotations provide meta-information about the program elements that they annotate. For example, the `@Deprecated` annotation indicates that the annotated program element is no longer supported [1]. Annotations are widely used in various types of Java applications including enterprise Java applications [12, 48]. Major Java enterprise frameworks, such as Spring [38] and MicroProfile [24], facilitate the development of enterprise applications mainly through annotations.

MicroProfile is a collection of specifications that provides Application Programming Interfaces (APIs) for client developers to create applications with a microservice architecture (*"small, autonomous services that work together"*) [24, 25]. For example, the MicroProfile Health specification provides mechanisms to check whether a service is started, ready to accept requests, or live [11]. MicroProfile provides these functionalities mainly through annotation-based APIs. For example, in Figure 1, the `@Liveness` annotation is used to check whether the authorization service is live [5]. There are different implementations of MicroProfile specifications such as Open Liberty [19], Helidon [34], and Payara [31].

Similar to how API calls have usage rules that determine correct behaviour (e.g., `hasNext()` must return `true` before invoking `next()` on an `Iterator` object to avoid throwing a `NoSuchElementException` [40]), annotations also have usage rules. For example, as shown in Figure 1, the target class of the `@Liveness` annotation must implement the `HealthCheck` interface to register for a liveness check [5]. Container management systems such as Kubernetes use liveness checks to see if a particular container needs to be restarted [7]. Violation of this usage rule will cause the liveness check to not function properly, without showing any explicit error message. We refer to such violations of annotation usage rules as *API misuses*, or *misuses* for short.

To prevent annotation misuses, we would ideally have access to checkable usage rules and tools that allow automated checking of

client code against these rules. There are existing tools that enable writing annotation usage rules and scanning a target codebase for misuses [8, 23, 49]. However, such tools assume that the usage rules are already known and readily available to encode, which is typically not the case [36, 43]. Additionally, manually creating API usage rules from scratch requires human effort, which can be difficult and time-consuming [18, 39].

To address the issues of writing API usage rules from scratch, researchers have utilized pattern mining techniques [10, 21, 26, 40, 46]. Pattern mining discovers usage rules in an automated fashion. The general idea is that if the frequency of a usage pattern is more than a user specified frequency, we consider that pattern as a rule [35] and violations of this pattern as a misuse. Researchers have mined different types of code artifacts, such as source code [26, 40, 46], API change history [21], and execution traces [10], to extract rules automatically. However, misuse detectors that directly use mined rules to detect misuses suffer from low precision, with the state-of-the-art detector having only 33% precision [40], which limits practical use. Reasons for the low precision include the fact that mined patterns may represent common usages (i.e., idioms) instead of rules. Mined patterns may also represent rules that are not entirely correct (i.e., partially correct rule). A partially correct rule might have some missing or extra elements.

Overall, manually writing API usage rules from scratch is time consuming but leads to more accurate rules; on the other hand, automatically mining patterns relieves the manual burden but can lead to inaccurate rules. In this paper, our goal is to generate annotation usage rules by combining the advantages of these two approaches, while mitigating their disadvantages. Our main idea is to introduce a human into the loop, but without the full burden of authoring rules from scratch. We use mined, unverified annotation usage rules (i.e., candidate rules) as starting points for human experts to create usage rules so that the process of creating rules will be less difficult and tedious. At the end of this process, we have human-validated annotation usage rules that can be directly used for detecting misuses or enriching documentation.

In this paper, we describe our industrial collaboration with IBM to create a human-in-the-loop approach for generating MicroProfile annotation usage rules. Figure 2 shows an overview of our approach. We first mine candidate rules from MicroProfile client projects (Step 1). We then present the mined candidate rules to human experts for validation (Step 2). Finally, we automatically generate static analysis checks from the confirmed rules; these checks are used by our misuse detector to find annotation misuses in MicroProfile client projects (Step 3). In our previous work, we explored using frequent-itemset mining to mine candidate MicroProfile annotation rules [28]. This paper thus focuses on Steps 2 and 3 of the process, which are essential to combine the two worlds of automated pattern mining and manual rule authoring.

Specifically, we focus on validating mined candidate rules and developing a misuse detector encoded with the validated rules. To validate mined candidate rules, we develop a web-based tool, Rule Validation Tool (RVT)[1]. RVT automatically encodes mined candidate rules in a domain-specific language (DSL) and presents them to experts for validation. RVT allows experts to not only validate the presented candidate rules, but also modify the ones that are partially correct. After experts validate or modify the candidate rules, RVT generates a final set of static analysis checks for the validated rules. We also develop a misuse detector in the form of a Maven plugin that uses the generated static analysis checks to detect annotation misuse in client projects. Finally, we perform a user study with four MicroProfile API experts from our industry partner to assess the usability of RVT for validating and modifying mined candidate rules. Our results show that API authors find that having a starting point for rule authoring is useful, and also find our rule format to be easily understandable. Our participants also provide us with additional feedback on how to improve RVT and its DSL for authoring rules, which we discuss in this paper.

[1]https://github.com/ualberta-smr/generating-annotation-usage-rules

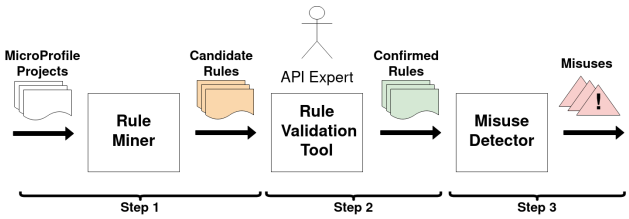

Figure 2: An overview of our human-in-the-loop rule generation approach.

```
1 @Counted(name="rate")
2 public int getCurrentRate() { return rate; }
```

Figure 3: An example illustrating the usage of the MicroProfile `@Counted` annotation [4].

2 Background

In this section, we first introduce our target framework, MicroProfile. We then introduce Java annotations and how the MicroProfile framework makes use of them. We also briefly explain our previously developed approach for mining MicroProfile annotation usage rules [28], since we used these mined rules as input to RVT.

2.1 MicroProfile

MicroProfile is a collection of specifications that provide client developers with all the necessary APIs to facilitate the development of microservices applications in Java [24]. Each MicroProfile specification targets a specific functionality such as Configuration, Health, and Metrics. MicroProfile provides these functionalities mostly through annotations. For example, the `@Counted` annotation from MicroProfile Metrics shown in Figure 3 tracks the number of times the annotated program elements (i.e., methods or constructors) get invoked [4]. Our industry partner is interested in ensuring proper usage of MicroProfile annotations by client developers.

2.2 Java Annotations

Java annotations provide meta-information about program elements (e.g., classes, fields, and methods) they annotate [3]. Developers use annotations for different purposes such as dependency injection, data binding, and code generation [48].

There are certain usage rules that client developers must follow when using annotations. We divide these rules into two categories: explicit and implicit usage rules. *Explicit usage rules* are those explicitly defined by the framework or library developers when they declare an annotation, and which a compiler can automatically check. For example, framework developers can restrict the types of program elements an annotation can be used on (e.g., fields or methods) [2]. The compiler would then complain if client developers use the annotation on a different program element.

On the other hand, *implicit usage rules* are associated with the way an annotation is used in combination with other program elements including other annotations, and are not checked by the compiler. For example, a class annotated with the `@Liveness` annotation must implement `HealthCheck` as shown in Figure 1. Not implementing `HealthCheck` will cause the `@Liveness` annotation to be ignored, without showing any explicit error message. In this work, we are interested in generating implicit usage rules for MicroProfile annotations to prevent their misuse.

2.3 Pattern mining

Pattern mining seeks to automatically extract usage patterns or rules [35]. In our case, a pattern refers to a way an annotation is used in combination with other program elements, including other annotations. The main premise behind pattern mining is that a frequent usage represents a usage rule. We refer to mined rules that are not validated by experts as *candidate rules*. In our previous work, we devised a frequent-itemset based pattern mining approach to mine candidate rules for MicroProfile annotations [28].

Frequent-itemset mining is a data mining technique that is mainly used for association rule learning to discover interesting relationships between items in large databases [32]. For example, given *milk*, *bread*, and *butter* items in a store, we may have observed the following purchases from five different customers: *{bread}, {milk, bread}, {milk, bread}, {milk, bread, butter},* and *{bread, butter}*. In this example, if we consider a purchase that occurs at least three times as a frequent purchase, we have *{bread}* and *{milk, bread}* as frequent purchases or *frequent itemsets*. These frequent itemsets can be then used to generate association rules. *Association rules* are relational rules of the form "If X, then Y", or more precisely $X \Longrightarrow Y$ where $X, Y \subseteq F$ and F is a frequent itemset. The "if" part is called antecedent, and the "then" part is called consequent. For the above frequent itemset *{milk, bread}*, we will have *milk* $\Longrightarrow$ *bread*, *bread* $\Longrightarrow$ *milk*, or both candidate rules.

Figure 4 shows an example candidate rule that our previous pattern-mining approach discovers [28]. Each item in a candidate rule represents a tuple $\{P_1, R, P_2\}$, where P_1 and P_2 are program elements, including annotations, and R is a relationship between the program elements. For example, for the item "`Class annotatedWith @ApplicationScoped`" in Figure 4, `Class` and `@ApplicationScoped` are program elements and `annotatedWith` is a relationship. In our previous work [28], we define the following eight different relationships for MicroProfile annotation usage based on our manual analysis of known annotation usage rules:

(1) `annotatedWith` represents what annotation a program element has been annotated with. A program element can be a class, field, method, constructor, and method and constructor parameters.
(2) `hasType` represents the data type of a field.
(3) `hasParam` represents a parameter of a method, constructor, or annotation.
(4) `hasReturnType` represents the return type of a method.
(5) `extends` represents a class extension.
(6) `implements` represents an interface implementation.
(7) `definedIn` represents an annotation parameter value that has been defined in `microprofile-config.properties`.
(8) `declaredInBeans` refers to the bean declaration of a class in the `beans.xml` file.

```
{
    "antecedent": [
        "Class annotatedWith @ApplicationScoped",
        "Class annotatedWith @Readiness"
    ],
    "consequent": [
        "Class implements HealthCheck"
    ]
}
```

Figure 4: A candidate rule mined by Nuryyev et al. [28].

Our mined candidate rules can have items with any of the above eight relationships. In our previous work, we showed that our mining technique is effective in discovering rules including new, previously unknown rules [28]. However, this evaluation was done *manually* with only one API expert who was one of our direct industry collaborators. In this paper, we specifically focus on the end-to-end streamlined process of providing the means for API experts to easily validate these mined candidate rules through proper tool support, as well as to automatically generate annotation usage rules that can be used in misuse detection.

3 Generating Annotation Usage Rules

In this section, we describe our approach to generate annotation usage rules and developing a misuse detector for MicroProfile. Figure 2 illustrates the overview of our approach. We first use our existing pattern-mining approach to mine candidate usage rules from MicroProfile client projects (Section 2.3). We then present the mined candidate rules to experts for validation, through RVT. Finally, we develop a misuse detector that uses the validated rules to detect misuses. In this section, we describe the specific contributions of this paper: the rule validation and misuse detection components.

3.1 Rule Validation Tool (RVT)

To facilitate rule validation, we develop a web-based Rule Validation Tool (RVT). Rule validation is a twofold process. Given mined candidate rules, RVT automatically encodes them in an easy-to-understand format. RVT then presents the encoded rules to experts for validation who can confirm a rule as is, confirm a rule after modifications, or reject a rule as invalid.

3.1.1 Rule Encoding. The candidate annotation usage rules that we mine are in a JSON [13] format as shown in Figure 4. Since

experts will read, modify, and validate the candidate rules, we need to present them in a format that is easy to comprehend. Overall, we do not want experts to spend too much time trying to learn and understand the rule format, because it defeats the purpose of reducing human effort in generating rules. The rule specification format should also support most of the mined relationships. We consider three options for selecting a rule encoding format: (1) use the original JSON format, (2) create a domain-specific language (DSL) from scratch specifically designed for our needs, and (3) use an existing DSL that supports our requirements. We discuss these options with our industry partner to understand how they perceive their pros and cons.

The original JSON format (option 1) is concise and simple to understand. However, the structure of the mined candidate rules, specifically items in both the antecedent and consequent, is ad-hoc at best. More importantly, the JSON rule format does not have a grammar, which means any post-processing of candidate rules will involve lots of string manipulations and regular expressions.

Creating a DSL from scratch (option 2) gives us the advantage of customizing it according to our requirements. However, creating a DSL from scratch is time-consuming, when compared to using an existing DSL. To present the mined candidate rules in a specific format, we considered multiple existing DSLs (option 3) that were originally developed for encoding various types of API usage rules. Specifically, we considered AnnaBot [8], RSL [49], CrySL [17], RulePad [23], ModelAn [27], and Smart Annotations [15].

Among the existing DSLs, we exclude ModelAn [27] and Smart Annotations [15], because they require modifications to the MicroProfile source code. CrySL [17], on the other hand, is designed for the specification of correct cryptography API uses in Java, and it heavily focuses on control and data-flow relationships, whereas annotation usage does not require control and data-flow relationships. Therefore, we also exclude CrySL from potential formats for presenting candidate rules. AnnaBot [8] and RSL [49] are specifically designed for writing annotation usage rules in a declarative way. However, they only support annotation usage rules between two annotations. For example, if `@X`, then `@Y`. They do not support usage rules between annotation and other program elements (e.g., field and method). For example, if `@X`, then a method must return `Z`. However, most of our mined candidate rules specify relationships between annotation and other program elements [28]. Therefore, we also exclude AnnaBot and RSL from potential formats for presenting candidate rules.

RulePad [23] is a tool that allows users to write code design rules (e.g., `function with type "void" must have name "set..."`) and check the source code for any violations of those rules. Developers can create rules using a semi-natural DSL created by the RulePad authors. For example, Figure 5 shows how the mined rule from Figure 4 can be expressed in RulePad's DSL.

Based on the feedback we received from our collaborators at IBM, we decide to use RulePad (option 3), because: (1) it has an intuitive English-like syntax, (2) rules have IF/THEN format that fits nicely with the mined rules, and (3) it has a grammar, making later extensions easier to implement. However, there were also three RulePad shortcomings that we needed to address.

First, RulePad does not support writing rules for the following relationships that appear in MicroProfile candidate rules: (1)

```
1 class with annotation "ApplicationScoped" and
      annotation "Readiness"
2 must have implementation of "HealthCheck"
```

Figure 5: An example illustrating the RulePad rule for the mined candidate rule shown in Figure 4.

method/constructor parameters having annotations, (2) annotations having parameters, and (3) configuration files. Therefore, we extend RulePad's grammar to support those relationships. Second, some RulePad keywords, specifically `declaration statement` and `function`, do not reflect Java language terminology. This might affect the readability or comprehensibility of rules. Therefore, we change the keywords to make them better align with Java language terminology. We change `declaration statement` to `field` and `function` to `method`.

Third, some of RulePad's syntax was perceived as too verbose. To address this, we introduce shortcuts into the DSL. We create a shortcut to express method, constructor, or annotation parameters. For example, a `String` parameter with the name `foo` is expressed as `parameter with type "String" and name "foo"` in RulePad's original DSL. We shorten it to `parameter "String foo"`, mirroring Java-style parameter declaration. We also create a shortcut that allows grouping of annotations from the same package. For example, to require one of JAX-RS HTTP method annotations (GET, POST, PUT, DELETE) [29], the corresponding RulePad expression is `annotation "javax.ws.rs.GET" or annotation "javax.ws.rs.POST" or annotation "javax.ws.rs.PUT" or annotation "javax.ws.rs.DELETE"`. We condense that expression into `annotation "javax.ws.rs.[GET|POST|PUT|DELETE]"`.

RVT takes the mined candidate rules in JSON format and converts them into RulePad rules, which we present to experts for validation through a Graphical User Interface (GUI) described next.

3.1.2 RVT's GUI. RVT provides a GUI for experts to go through and validate the presented candidate rules. Figure 6 shows the main elements of RVT's GUI. The *Rule Authoring Editor* ① presents the candidate rule, encoded in RulePad format, that needs to be validated. To improve the readability of the presented rules, we equip the editor with syntax highlighting and formatting features. The *Code Preview* ② provides a minimal Java code example to show what the presented candidate rule corresponds to in actual Java code. That preview also highlights the code representing the antecedent (orange) and the consequent (green), enabling visual separation between the two parts of a candidate rule. The goal of the Code Preview is helping experts further understand the candidate rule presented in the Rule Authoring Editor. The *progress indicator* ③ shows the total number of candidate rules that need to be validated, the position of the currently presented rule among all the candidate rules, and the number of candidate rules left to validate. RVT has two *labeling buttons* ④ ("Confirm rule" and "Not a rule") to label the presented candidate rule as a correct or incorrect rule. RVT also has two *rule navigator* buttons ⑤ to navigate through candidate rules. Finally, the question mark ⑥ indicates *help* and opens a tutorial page on using RVT. The tutorial page contains information about the RulePad grammar and all possible actions that experts

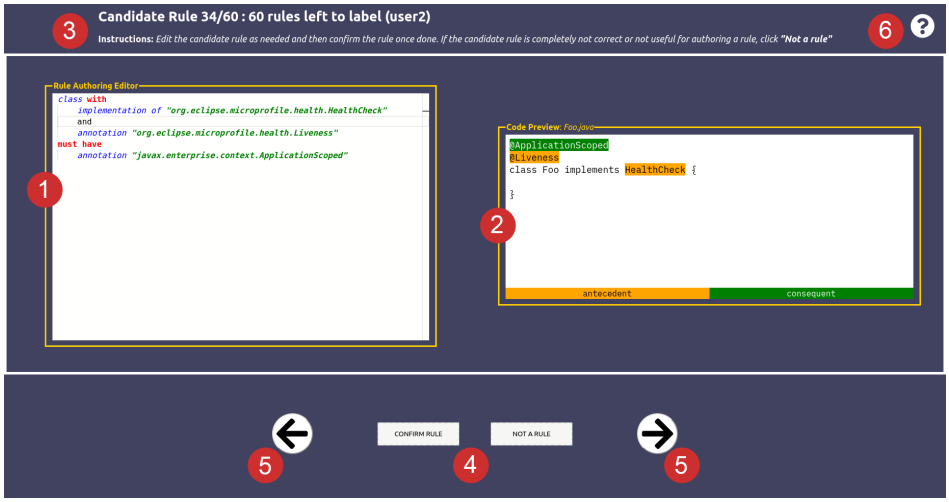

Figure 6: The main GUI elements of our Rule Validation Tool (RVT). Features 1–6 are detailed in Section 3.1.

can perform using RVT. Overall, RVT's GUI not only presents the encoded rules but also provides the necessary features to help experts understand the presented rules.

3.1.3 Rule Validation Process. Once RVT presents a candidate rule, experts can take one of the following three actions to validate and label the rule:

- *Confirm the rule as-is*: If the presented candidate rule is correct, experts can label the rule as correct by clicking "Confirm rule".
- *Confirm the rule with changes*: If the presented candidate rule is a partially correct rule, e.g., the rule has some missing or extra items, experts can edit the presented rule by adding, removing, or modifying items of the rule using the Rule Authoring Editor. After finishing their editing, they can confirm the edited rule by clicking "Confirm rule".
- *Label the rule as incorrect*: If the presented candidate rule does not represent any annotation usage rule, experts can discard this rule by clicking "Not a rule".

For each presented candidate rule, RVT stores the validated form of the rule and the label in a database. Once experts validate all candidate rules, we use only confirmed rules for misuse detection.

3.2 Misuse Detector

Since our industry partner is interested in ensuring the correct usage of MicroProfile annotations by client developers, we next focus on developing a misuse detector that uses the correct rules from the previous step to detect annotation misuses.

We first consider using RulePad for misuse detection as it comes with a misuse detector out of the box. However, we find that RulePad's misuse detector is not suitable for our needs. First, RulePad's misuse detector works in a browser. It has an IDE plugin that sends client code to the web UI for misuse detection. To use RulePad's misuse detector, both the web UI and the IDE plugin need to run simultaneously. However, we are interested in using the detector as a standalone tool. Second, RulePad's misuse detector transforms Java code into XML representations, which do not contain resolved types. Third, the XML representations do not represent some of the Java code constructs such as the data type of annotation parameters. We could have resolved some of the issues by introducing a preprocessing step before RulePad transforms Java code into XML representations. However, adding such a preprocessing step would have negatively affected the performance of RulePad's misuse detector. Instead, we build our own misuse detector as follows.

A misuse detector may be implemented as a build tool (Maven [22] or Gradle [42]) plugin or an IDE plugin. Client developers can use a build plugin as a part of their continuous integration (CI) pipeline. However, locally, client developers have to explicitly run a build plugin to detect misuses. Unlike build plugins, an IDE plugin would detect misuses instantly as developers are working on their code. However, an IDE plugin cannot be used in the CI pipeline. After a discussion with our industry collaborator, we decided to build our misuse detector as a Maven plugin to enable CI integration.

Our misuse detector uses the confirmed rules to generate static analysis checks. We use JavaParser [30] to parse the Java files of the target project and to resolve types. Our static analysis checks are also encoded using JavaParser. We automatically extract the antecedent part of a RulePad rule and search for occurrences of it in a given Java file's Abstract Syntax Tree (AST). If we find an occurrence of the antecedent, we then check if the consequent holds. If not, then we have detected a misuse. The detector scans one Java file at a time of the target project for any misuse of the available rules. Once the detector scans all the Java files of the target project, the detector prints a detailed report for each misuse it finds in the target client project. The report contains the misused rule, the misuse location, and the missing element. Figure 7 shows an example of a report generated by our misuse detector.

To verify that our misuse detector works as expected, we evaluate it against the set of 16 open-source and proprietary MicroProfile client projects with known misuses from our previous work [28]. In that work, we detected 100 misuses of five distinct usage rules. We encode these five rules in our extended RulePad format and input them to our misuse detector, which automatically converts them into static analysis checks to detect misuses. In contrast, misuse

```
$ mvn violation-detector: scan
[WARN] For rule: QueryGraphQLAPIRule
[WARN]    class with function with annotation "Query" \
[WARN]       must have annotation "GraphQLApi"
[WARN]    Class FooBar is missing the following element(s):
[WARN]       [@org.eclipse.microprofile.graphql.GraphQLApi]
[WARN]     Location: (line 22, col 1) - (line 77, col 1)
```

Figure 7: A sample report generated by our misuse detector.

checks from our previous work are entirely written manually. Our misuse detector successfully detects all previously identified 100 misuses, demonstrating that our static analysis check generation works correctly.

4 User study

To evaluate the usefulness of RVT in modifying and validating the mined candidate rules, as well as the usefulness of the whole idea of using mined rules as starting points, we conduct a user study with MicroProfile API experts. Our goal is to answer the following three research questions:

RQ1. Is the rule specification DSL in RVT expressive enough for specifying rules? We adopt RulePad with some extensions as our DSL of choice for encoding rules. There is a possibility that an API expert might want to specify a constraint that cannot be expressed using RulePad. Thus, we want to know how expressive our extended RulePad DSL is for authoring annotation-based API usage rules.

RQ2. Is RVT useful for the modification and validation of mined rules? The key concept in our proposed pipeline is having a human in the loop. Thus, we want to know if RVT makes it easy for experts to author and validate the mined rules.

RQ3. Are candidate rules effective in alleviating the difficulties of writing API usage rules from scratch? We want to understand if the mined candidate rules provide good starting points for API experts when authoring rules. Overall, we want to determine if the idea of having mined rules as a starting point is useful to API experts.

4.1 Experiment Setup

The experiment is an online, 90-minute Zoom session where experts use RVT to validate the presented candidate rules. We audio and video record the session, with participants' consent and after our university's ethics clearance, for post-analysis purposes. The experiment is divided into three parts that we describe below.

4.1.1 Tutorial and setup. At the beginning of the experiment (up to 30 minutes), experts go through a tutorial that we prepared to get familiar with RVT and the DSL that we use to present rules (i.e., the extended RulePad).

4.1.2 Live experiment. After participants get familiar with RVT, we proceed to the main experiment task, where API experts validate candidate rules encoded in the RulePad DSL. For each candidate rule, we first ask the participants to rate it in terms of understandability of the presented rule on a scale of 1 to 3 (1-hard to understand, 2-neither hard nor easy to understand, 3-easy to understand). This task enables us to quantify how easy it is for API experts to understand a given rule and contributes to the evaluation of RQ1. After getting familiar with the presented rule, participants proceed to validate it. Participants are allowed to use online resources such as documentation and online discussion forums, if needed. To validate a rule, participants can (1) confirm the rule as is, (2) confirm the rule with changes, or (3) reject the rule. During this validation process, we employ the think-aloud protocol [44] where we ask participants to verbally share the reasoning behind their decisions. For example, when a participant rejects a candidate rule, we ask them to share the reasons that led them to this decision. This feedback can help us improve the mining process.

Table 1: Number of candidate rules mined for each MicroProfile specification [28]. One rule belongs to both GraphQL and OpenAPI specifications, hence the total is 23, not 24.

MicroProfile Specification	# mined rules	# confirmed rules
Config	1	0
GraphQL	3	N/A
Health	3	3
JWT-Auth	2	1
Metrics	4	3
OpenAPI	6	2
REST Client	3	N/A
Reactive Messaging	2	2
Total	**23**	**11**

4.1.3 Exit survey. At the end of the session, we ask participants three rating-based (RB) and three open-ended (OE) questions. For the rating-based questions, participants can also provide verbal explanations for their ratings. We ask the following questions:

RB1: For rule authoring, having an existing candidate rule as a starting point is easier than writing a rule from scratch (strongly disagree, disagree, neither agree nor disagree, agree, strongly agree). This question addresses RQ3.

RB2: Having a dedicated tool for rule validation makes it easy to validate rules (from strongly disagree to strongly agree). This question addresses RQ2.

RB3: How do you rate the difficulty level of editing rules using RVT? (very hard to edit, hard to edit, neutral, easy to edit, very easy to edit). This question addresses RQ2.

OE1: Are there additional code constructs you think need to be a part of RulePad? This addresses RQ1.

OE2: What types of additional information could have assisted you in validating the rules? This indirectly addresses RQ2 and enables us to know what other information experts would find helpful.

OE3: Are there any additional rules you can think of that were not presented? This question does not address a specific RQ but enables us to understand what rules the mining process cannot discover and what other code relationships need to be tracked (which may require further RulePad extensions).

Our open-ended questions allow participants to share valuable feedback with us, which helps us further improve our approach.

4.2 Participant Recruitment

MicroProfile API experts (i.e., direct contributors to various MicroProfile specifications) are the target population of our study. We

drafted a recruitment email that our industry collaborator sent to six MicroProfile API developers in IBM. Our goal was to recruit at least one expert for each MicroProfile specification that we have mined rules for. Table 1 shows how many candidate rules we mined for each MicroProfile specification. Four API experts (P1-4) agreed to participate in the user study. Before the experiment, for each participant, we collected background information on which MicroProfile components they are familiar with and created a set of candidate rules that contain APIs from these components. P1-4 are MicroProfile contributors, working in their current teams for 4, 6, 5 and 2 years, respectively. P1 is responsible for OpenAPI, Reactive Messaging, and Config components, P2 is responsible for Health, P3 is responsible for Metrics, and P4 is responsible for JWT-Auth. For the study, the participants P1-4 validated 9, 3, 4 and 2 candidate rules, respectively.

4.3 Results

Participants validated a total of 18 rules, and there were no common rules shared between participants due to their different expertise. P1-4 confirm 4/9 rules, 3/3 rules, 3/4 rules, and 1/2 rules respectively. Overall, participants label 11/23 rules as correct, with all except one rule (from MicroProfile Metrics) requiring modifications. All presented rules for MicroProfile Health and Reactive Messaging are partially correct. For MicroProfile Metrics, P3 considers the only rejected rule as "best practice" and not necessarily incorrect.

We now present the main results of our user study, where we focus on the whole pipeline rather than the accuracy of the mining process, which we evaluated in our previous work [28].

RQ1: Expressiveness of the extended RulePad DSL in RVT. Figure 8 shows how participants perceive the presented candidate rules in terms of their understandability. The graph shows that the majority of the presented candidate rules are easy to understand for participants. P2 mentions that the English-like syntax of RulePad makes it easy to learn in a short period of time. Recall that we introduced two constructs to RulePad to reduce the verbosity of the DSL (Section 3.1.1). We observe that P1 and P2 use the shortcut that allows grouping of annotations from the same package, showing the usefulness of the shortcut. We find that while the extended RulePad is expressive enough to specify most of the code constructs needed to encode annotation usage rules, there is still room for improvement. Participants suggest that the extended RulePad can be further improved by including the following code constructs:

(1) Specify mutual exclusivity. A rule might require usage of only one annotation from a set of annotations. Currently, the extended RulePad supports disjunctions (i.e., or) which does not guarantee mutual exclusivity (i.e., xor).
(2) Invert a predicate. Our extension to RulePad does not support negations. For example, our extension cannot encode the following hypothetical rule: a field with annotation A and not with annotation B requires annotation C.
(3) Require overriding a specific method. Method overriding is useful in two cases. First, it enables rule completeness. Currently, we can specify that a class needs to implement an interface, but a complete rule must indicate which method needs to be overridden/implemented from that interface. Second, given a predicate, an expert might want to require overriding specific methods (e.g., if class is annotated with X and extends Y, then it must override method Z).
(4) Shortcuts for frequently used MicroProfile constructs such as CDI beans, JAX-RS resource methods (i.e. a method that is annotated with request method designators such as `@GET` or `@POST`) or classes (i.e., a class that either is annotated with `@Path` or contains at least one resource method). For example, instead of saying `method with annotation "Operation" must have annotation "[GET|POST|PUT|DELETE|..]"`, an expert can simplify the rule to `method with annotation "Operation" must be a JAXRS resource`.

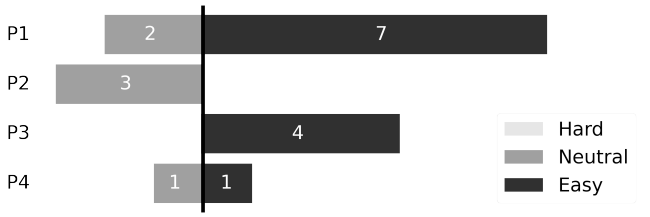

Figure 8: Understandability of candidate rules

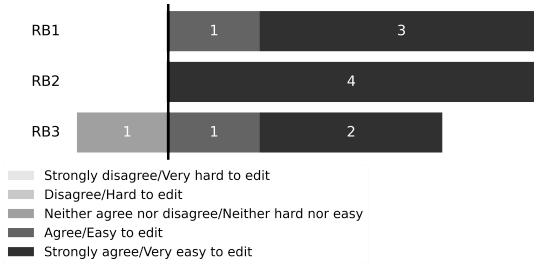

Figure 9: RB1 results regarding having a starting point for rule authoring. RB2 results regarding usefulness of having a dedicated rule validation tool. RB3 results for levels of difficulty of editing rules using RVT.

RQ2: Usefulness of RVT in modifying and validating candidate rules. Figure 9-RB2 shows what participants unanimously agree that RVT is useful. Therefore, we conclude that having a dedicated tool for rule validation is useful for API experts. That said, Figure 9-RB3 shows that our participants have varying opinions on the level of difficulty of editing rules using RVT. P1 who rated "Neither hard nor easy" states that in some cases the rules written in the extended RulePad format are not how they would be written as a specification. For example, consider the following rule: "a field with the `@ConfigProperty` annotation should have the `@Inject` annotation unless the class has the `@ConfigProperties` annotation". In the extended RulePad, we encode the rule as "`class with field with annotation "ConfigProperty" must have annotation "ConfigProperties" or field with annotation "Inject"`". While this rule is correct, P1 argues that it is not natural to write the rule in that format. Instead, P1 mentions that an API expert would write the rule as "`field with annotation "ConfigProperty" must have annotation "Inject" unless enclosing class`

`has annotation "ConfigProperties"`". This aligns with the previous suggestion of supporting predicate negation in RulePad.

We now discuss what other information participants think can help them in the rule validation process. From the existing assisting components of RVT, participants find the code preview section particularly helpful for visualizing how a rule might potentially look. P3 states that the rules being presented pre-formatted makes rules easier to understand. Participants share the following ideas for potential enhancements:

- Easier access to Java documentation. Finding the correct documentation page may take time, and providing quick access to it can be useful.
- Auto-completing fully qualified names (FQNs). When generating static analysis checks, we expect the validated rules to have FQNs. While candidate rules have FQNs, it might be hard for experts to know the FQN for every annotation/class they need to add or modify. Providing such assistance can reduce the time spent on looking for the correct package name.
- Syntax checking for the DSL. Participants believe that having a syntax error checker can assist them in writing rules properly.

RQ3: Effectiveness of the mined rules in alleviating the difficulties of writing usage rules. Figure 9-RB1 illustrates what participants think about having starting points for authoring rules. Our results show that having starting points is helpful. According to P1, given a rule, it is generally easier to point out the problems with the rule. In a similar fashion, P3 states that as with anything in tech, it becomes easier once you have something to work with. In response to OE3, P1 states that there are probably additional rules to encode; however, to find them, one needs to go through the documentation. Note that this statement strengthens our argument that having starting points reduces the effort of manually going through documentation to find rules, especially that not all API usage rules are documented.

One of the potential issues with finding API usage rules through pattern mining is that a mined pattern can contain deprecated APIs. There was one such rule that P2 had to validate. While converting the candidate rule into the correct rule, P2 explains that the candidate rule uses the `@Health` annotation which has been deprecated since MicroProfile Health version 2.0 and is no longer part of the API [9]. This case shows that having human validation is critical when it comes to mining rules.

4.4 Threats to Validity

Internal validity. Our user study focuses on API experts' experience and perception of the rule validation process. Our pipeline also includes the last step where we generate static analysis checks from the validated rules, which we use for misuse detection. While this step is not evaluated in the user study, we verify that it works correctly by ensuring that we can detect all previously known misuses in 16 projects. However, we do not analyze any additional unknown misuses that the detector finds.

Construct validity. To mine candidate rules, our pipeline uses our previously developed and evaluated pattern mining technique from our previous work [28]. The quality of the mining process can affect the overall experience of participants in our user study. However, the user study does not focus on measuring the correctness of the rules but rather focuses on the editing and validation process.

External validity. Our goal was to validate all the mined rules by at least one API expert. We reached out to six of the relevant MicroProfile API experts working for our industry partner, and four agreed to participate. These four participants validated 18 out of 23 mined rules (78%) from five different MicroProfile specifications. Although we could not validate all the mined rules, the validated rules cover most of the annotation relationships in our rules. Our work focuses on validating MicroProfile annotation usage rules. While, in principle, our approach can be applied to other annotation-based libraries, we present only a case study of MicroProfile and our findings may not generalize beyond that. Future work can reuse our pipeline to investigate its applicability to other libraries and frameworks.

5 Discussion

In this paper, we proposed a human-in-the-loop approach to generate annotation usage rules for MicroProfile. We developed a web-based tool, RVT, to facilitate rule validation and generation. To evaluate the usefulness of RVT and our approach, we performed a user study with MicroProfile API experts. We now discuss the implications of our findings.

1) Generating API usage rules. Our main objective in this paper is to generate accurate annotation usage rules while reducing the burden of writing them from scratch. Therefore, we use a pattern mining technique to automatically mine candidate rules that provide starting points to experts for generating accurate rules. The results of our user study (RQ3) show that all the API experts agree or strongly agree that having starting points in the form of mined candidate rules reduces not only the difficulty but also the effort of writing rules. They state that it is easier and takes less effort to work with candidate rules and find problems in the rules rather than discovering a rule manually. Our participants also confirm 10 of the 16 presented candidate rules and modify nine of the 10 confirmed rules, which indicates that most of the presented candidate rules are partially correct or incorrect rules. Thus, using the mined candidate rules directly for misuse detection could have produced a lot of false positives. Therefore, our approach that introduces experts for validating the mined rules is critical for generating accurate rules. The results also show that experts go to extra lengths to produce accurate rules. For example, P2 checked whether the generated rules use any deprecated APIs.

Our results show that 10/11 confirmed rules are present in the documentation. This shows that our approach is effective in reducing the time spent on finding the rules in the documentation. While these rules are present in the documentation, they are not automatically checked for, and client developers might violate them. However, the remaining one rule is not documented, showing that validated rules can also be used for improving the documentation.

2) Facilitating rule validation. To facilitate rule validation, our approach uses an extended version of the RulePad DSL. Not only can experts use RVT to validate the presented rules, but they can also modify them. Therefore, it is critical that the DSL is able to express or construct annotation usage rules. While our results (RQ1 and RQ2) show that the DSL does not lack any grammar to express the confirmed rules, participants would like the ability to express

API usage rules in a more natural form and with finer granularity. For example, P1 states that RulePad's DSL might not be the most natural way to express a rule in some cases. However, expressing a rule in a natural form can be challenging, because it might be hard to find a consistent DSL format or syntax that allows natural expression. Addressing this limitation requires another user study where the focus is on the way experts naturally author rules without adhering to a specific format. We can then analyze the produced rules and see if there are common patterns in the rules that can be incorporated into RulePad grammar. Additionally, introducing more logical operators, such as XOR, NOT, and NOR, would allow experts to express rules with finer granularity.

Another avenue for improvement is the assisting features in RVT. Currently, the feature that has been brought up the most is the auto-completion of fully qualified names. This feature will allow experts to easily specify fully qualified names without consulting documentation, which will improve the overall user experience. We can implement the auto-completion feature by extracting fully qualified names of MicroProfile APIs from the documentation and storing them in a database integrated with our Rule Authoring Editor. We can also provide easy access to Java documentation for all the program elements used in the rule. This feature can be a part of the Code Preview where clicking on a program element will open the corresponding Javadoc page. Finally, we can provide another labeling button to allow experts to label a rule as a "best practice". When used for misuse detection, the best practice rules can produce warnings instead of errors. API experts also suggest including a mechanism to check for syntax errors in the DSL.

6 Related Work

6.1 Mining API Usage Rules

Researchers have proposed various pattern mining techniques to automatically extract API usage rules [14, 26, 35, 41, 45, 46, 50]. However, most of the pattern mining solutions focus on control and data-flow relationships [26, 41, 45, 46], with little focus on annotation usage. In general, annotation usage does not depend on control and data-flow analysis.

To the best of our knowledge, with the exception of our previous work [28] discussed in Section 2.3, there is no existing work on mining annotation usage rules. However, Liu et al. [20] use deep learning to train an annotation prediction model, DeepAnna. The model learns from the structural (abstract syntax tree) and the textual (i.e. annotation name, method name, method contents) context of the source code to recommend annotations and detect their misuse. However, DeepAnna cannot specify reasons behind its recommendations. For example, DeepAnna might suggest to use the annotation `@Liveness` on a class (consequent), but it cannot specify the reason behind that suggestion (antecedent). We need to have both an antecedent and a consequent to create an annotation usage rule. Moreover, DeepAnna focuses only on recommending class and method annotations, while our work considers the representation of annotation usage rules that involve fields, parameters and configuration files in addition to classes and methods, and the detection of misuses of such rules.

Pattern mining can produce rules that are insecure, overly simplistic, or missing context [17]. Pattern mining techniques may also mine usage rules for deprecated APIs, which are not useful. Thus, it is critical to validate or correct the mined rules to transform them into valid API usage rules.

6.2 Writing API Usage Rules

There is an abundance of rule authoring tools to write API usage rules [8, 17, 23, 49]. Most of these tools have their own DSL, and they often target specific libraries or domains. For example, CrySL [17] is designed for writing Java Cryptography API (JCA) usage rules. It focuses on control and data-flow relationships between different method calls and does not support annotations.

There are also tools or DSLs that focus specifically on writing usage rules for annotations. RSL [49] and AnnaBot [8] allow writing annotation usage rules in a declarative way. These tools support logical and aggregate operations such as "AND", "OR", "NOT", "at most one" or "for all". However, they do not support most of the relationships that our mining process produces. For example, AnnaBot does not support writing rules that specify a relationship between an annotation and a method return type. RSL, on the other hand, only allows writing rules that check the existence of an annotation on a program element.

Similar to previous DSLs, RulePad [23] allows software developers to create design rules and checks for misuses of these rules. Unlike previous tools, RulePad focuses on creating checkable and up-to-date documentation. Since the documentation is meant to be read by developers, to make the design rules easy to understand, RulePad provides an English-like DSL to encode rules in. RulePad's DSL has an IF/THEN structure which coincides with our mined rules. It also supports most of the mined relationships. Therefore, we use RulePad's DSL in this work with some customization to validate annotation usage rules.

Unlike focusing on usage of APIs from specific libraries or domains, there are tools, such as PMD [33], SpotBugs [37] and CheckStyle [6], that focus on general-purpose static analysis. These tools can be utilized to encode annotation usage rules. For writing custom rules in these tools, developers need to use either a general-purpose language such as Java or a querying language such as XPath (in the case of PMD)[47]. However, prior work shows that DSLs are easier to learn, read, and write than general-purpose languages [16]. Since we want to make rule validation as intuitive as possible, we did not choose a general-purpose static analysis tool for rule encoding.

The previous tools all provide external DSLs to encode the rules. There has also been *meta-annotation* solutions that encode rules in the source code directly [15, 27]. These solutions provide a set of meta-annotations (i.e. an annotation that can be applied to other annotations) to embed the usage rules in annotation source code. Since such techniques require source code modifications which would force us to modify the source code every time a new rule is discovered, these solutions are not useful for our use case.

All the above tools assume that the rules are readily available. However, someone needs to find the rules and encode them from scratch. For example, Krüger et al. [17] went through all the JCA documentation, and manually authored all the found rules in CrySL. There are two issues with this approach: (1) it is time-consuming, and (2) authors will miss undocumented rules. Our hybrid approach leverages pattern mining to reduce the time spent on authoring rules from scratch. It can also discover undocumented patterns.

7 Conclusion

We introduce a human-in-the-loop approach for producing accurate annotation usage rules of MicroProfile APIs. We leverage pattern mining to produce starting points for writing API usage rules. We build a specialized tool, RVT, to facilitate the rule validation process, and a misuse detector to automatically generate static analysis checks from correct rules. To make rules easily understandable, RVT extends an English-like DSL called RulePad for encoding mined rules. We evaluate our approach in a user study with MicroProfile subject matter experts. The user study results show that having starting points makes writing rules easier and that our proposed pipeline can be used to automatically produce accurate MicroProfile API usage rules. These usage rules can be integrated into static analysis tools to help MicroProfile client developers write less buggy code, or they can improve the documentation.

References

[1] Deprecated (Java SE 17 & JDK 17). 2021. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Deprecated.html.

[2] Target (Java SE 17 & JDK 17). 2021. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/annotation/Target.html. [Last accessed: May 31, 2022].

[3] Annotations. 2014. https://docs.oracle.com/javase/8/docs/technotes/guides/language/annotations.html. [Last accessed: May 31, 2022].

[4] Counted (MicroProfile 2.0.1-SNAPSHOT API). 2018. https://download.eclipse.org/microprofile/microprofile-2.0-javadocs-test/apidocs/org/eclipse/microprofile/metrics/annotation/Counted.html. [Last accessed: May 31, 2022].

[5] MicroProfile Health#Liveness check. 2021. https://download.eclipse.org/microprofile/microprofile-health-4.0/microprofile-health-spec-4.0.html#_liveness_check. [Last accessed: May 31, 2022].

[6] CheckStyle. 2022. https://checkstyle.org. [Last accessed: May 31, 2022].

[7] Readiness Configure Liveness and Startup Probes. 2022. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes. [Last accessed: May 31, 2022].

[8] Ian Darwin. 2009. Annabot: A static verifier for java annotation usage. *Advances in Software Engineering* 2010 (2009).

[9] MicroProfile Health Check 2.0 final. 2019. https://github.com/eclipse/microprofile-health/releases/tag/2.0. [Last accessed: May 31, 2022].

[10] Mark Gabel and Zhendong Su. 2008. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 339–349.

[11] MicroProfile Health. 2021. https://download.eclipse.org/microprofile/microprofile-health-4.0/microprofile-health-spec-4.0.html. [Last accessed: May 31, 2022].

[12] Ajay Kumar Jha and Sarah Nadi. 2020. Annotation practices in Android apps. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 132–142.

[13] JSON. 2009. https://www.json.org. [Last accessed: May 31, 2022].

[14] Hong Jin Kang and David Lo. 2021. Active learning of discriminative subgraph patterns for API misuse detection. *IEEE Transactions on Software Engineering* (2021).

[15] Andy Kellens, Carlos Noguera, Kris De Schutter, Coen De Roover, and Theo D'Hondt. 2010. Co-evolving annotations and source code through smart annotations. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 117–126.

[16] Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Varanda João Maria Pereira, Matej Črepinšek, Cruz Daniela Da, and Rangel Pedro Henriques. 2010. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems* 7, 2 (2010), 247–264.

[17] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2019. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2382–2400.

[18] Tien-Duy B Le, Lingfeng Bao, and David Lo. 2018. DSM: a specification mining tool using recurrent neural network based language model. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 896–899.

[19] Open Liberty. 2021. https://openliberty.io. [Last accessed: May 31, 2022].

[20] Yi Liu, Yadong Yan, Chaofeng Sha, Xin Peng, Bihuan Chen, and Chong Wang. 2022. DeepAnna: Deep Learning based Java Annotation Recommendation and Misuse Detection. *29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2022).

[21] Benjamin Livshits and Thomas Zimmermann. 2005. Dynamine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 296–305.

[22] Apache Maven. 2022. https://maven.apache.org. [Last accessed: May 31, 2022].

[23] Sahar Mehrpour, Thomas D LaToza, and Hamed Sarvari. 2020. RulePad: interactive authoring of checkable design rules. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 386–397.

[24] MicroProfile. 2022. https://microprofile.io. [Last accessed: May 31, 2022].

[25] Sam Newman. 2021. *Building microservices.* " O'Reilly Media, Inc.".

[26] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*. 383–392.

[27] Carlos Noguera and Laurence Duchien. 2008. Annotation framework validation using domain models. In *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 48–62.

[28] Batyr Nuryyev, Ajay Kumar Jha, Sarah Nadi, Yee-Kang Chang, Emily Jiang, and Vijay Sundaresan. 2022. Mining Annotation Usage Rules: A Case Study with MicroProfile. In *2022 38th International Conference on Software Maintenance and Evolution*. IEEE.

[29] JAX-RS package. 2018. https://javadoc.io/doc/javax.ws.rs/javax.ws.rs-api/2.1.1/javax/ws/rs/package-summary.html. [Last accessed: May 31, 2022].

[30] Java Parser. 2019. https://javaparser.org. [Last accessed: May 31, 2022].

[31] Payara. 2022. https://www.payara.fish. [Last accessed: May 31, 2022].

[32] Gregory Piatetsky-Shapiro. 1991. Discovery, analysis, and presentation of strong rules. *Knowledge discovery in databases* (1991), 229–238.

[33] PMD. 2022. pmd.github.io. [Last accessed: May 31, 2022].

[34] Helidon Project. 2021. https://helidon.io. [Last accessed: May 31, 2022].

[35] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2012. Automated API property inference techniques. *IEEE Transactions on Software Engineering* 39, 5 (2012), 613–637.

[36] Mohamed Aymen Saied, Houari Sahraoui, and Bruno Dufour. 2015. An observational study on api usage constraints and their documentation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 33–42.

[37] SpotBugs. 2021. https://spotbugs.github.io. [Last accessed: May 31, 2022].

[38] Spring. 2022. https://spring.io. [Last accessed: May 31, 2022].

[39] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. 2020. Extracting taint specifications for javascript libraries. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 198–209.

[40] Amann Sven, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. Investigating Next Steps in Static API-Misuse Detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 265–275. https://doi.org/10.1109/MSR.2019.00053

[41] Amann Sven, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. Investigating Next Steps in Static API-Misuse Detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 265–275. https://doi.org/10.1109/MSR.2019.00053

[42] Gradle Build Tool. 2022. https://gradle.org. [Last accessed: May 31, 2022].

[43] Gias Uddin, Foutse Khomh, and Chanchal K Roy. 2019. Towards crowd-sourced API documentation. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 310–311.

[44] Maarten Van Someren, Yvonne F Barnard, and J Sandberg. 1994. The think aloud method: a practical approach to modelling cognitive. (1994).

[45] Andrzej Wasylkowski and Andreas Zeller. 2011. Mining temporal specifications from object usage. *Automated Software Engineering* 18, 3 (2011), 263–292.

[46] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 35–44.

[47] XPath. 2022. https://developer.mozilla.org/en-US/docs/Web/XPath. [Last accessed: May 31, 2022].

[48] Zhongxing Yu, Chenggang Bai, Lionel Seinturier, and Martin Monperrus. 2018. Characterizing the Usage and Impact of Java Annotations Over 1000+ Projects. *arXiv preprint arXiv:1805.01965* (2018).

[49] Yaxuan Zhang. 2021. *Checking metadata usage for enterprise applications.* Ph. D. Dissertation. Virginia Tech.

[50] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*. Springer, 318–343.